Part 2

The Evaluation of Programs

Structure vs. Evaluation

- The data model describes the structure

```
23 + 4.5  ⟶  BinOp('+', Integer(23), Float(4.5))
```

- Evaluation is about how a program executes
- <u>Semantics</u>

Your Intuitions

- You, as a programmer, have certain intuitions about how programs actually work
- At least I hope so...
- We'll start with that
- Will dig deeper later

Literal Values

- Literals

```
2
2.3
'c'
```

- Literals just "are"
- They don't do anything other than exist.

Expressions

- Example: A binary operator (+)

```
left + right
```

- You evaluate each side first, then add together

```
(2*3) + (4*5)

      6 + (4*5)

      6 + 20

        26
```

- It's a recursive process ("show your work!")
- The final result is a value.

Names/Variables

- Names refer to objects in an environment

```
const pi = 3.14159;
var r float = 2.0;
var a = pi * r * r;
```

```
{
    'pi': 3.14159,
    'r': 2.0,
    'a': 12.56636
}
```

- An environment is a place to store things
- Two core operations: load/store

```
a = 2.0 * pi;      // Loads from "pi". Stores to "a"
```

Statements

- Statements execute one after another

```
result = result * n;
n = n - 1;
print result;
```

- Each statement causes some kind of change in the environment (note: includes I/O).
- "Imperative programming"

Conditionals

- if-statement presents two evaluation routes

```
if a < b {
   max = b;
} else {
   max = a;
}
```

- You evaluate the test first (a < b)
- Then, <u>only one</u> branch executes

Loops

- Repeated evaluation of statements

```
while n > 0 {
   result = result * n;
   n = n - 1;
}
```

- You evaluate the test first (n > 0)
- If true, evaluate the body and repeat.

Functions

- Consider a function

```
func sum_squares(x int, y int) int {
    return x*x + y*y;
}
```

- You evaluate arguments first. Then the body

```
sum_squares(2+3, 4+5)

sum_squares(5, 4+5)

sum_squares(5, 9)

          5*5 + 9*9
           25 + 9*9
           25 + 81
             106
```

Note: This is not the only way to do it, but most "normal" programming languages work like this.

"Applicative Order"

- Terminology: "Function Application"

Scoping

- Environments are nested/linked

```
const pi = 3.14159;

func area(r float) float {
    a = pi * r * r;
    return a;
}

print area(4.0);
```

Functions can see variables defined in the surrounding definition context.

(lexical scoping)

Globals

```
{
  'pi': 3.14159,
  'area': <function>
}
```

Locals (area)

```
{
  'r': 4.0,
  'a': 50.26544
}
```

Structures

- A structure is a container of data

```
struct Point {
   x int;
   y int;
}

p = Point(2, 3);
```

```
{
   'x': 2,
   'y': 3
}
```

- An instance holds values for all fields
- They're all together in one object
- Precise representation details vary.

Moving Beyond Intuition

- Yes, we have intuitions about how things "work" when we write programs
- Question: How do you turn this into a more formal specification?
- To write a compiler, you need a precise definition of how everything actually works.
- At a fine level of detail (i.e., language lawyer).

Formalizing Semantics

- One approach : Write an interpreter
- Example: Write a program that takes the data model and actually executes it.
- Sole focus: "What does the program do?"
- Sometimes known as a "definitional interpreter."

Definitional Interpreter

```
class BinOp(Expression):
    def __init__(self, op, left, right):
        self.op = op
        self.left = left
        self.right = right
```

(Model)

```
def interpret_binop(node, env):
    leftval = interpret(node.left, env)
    rightval = interpret(node.right, env)
    assert type(leftval) == type(rightval)
    if node.op == '+':
        return leftval + rightval
    elif node.op == '*':
        return leftval * rightval
    ...
```

(Interpreter)

Operational Semantics

- Writing a "definitional interpreter" is an approach taken by language designers and compiler writers in the real world
- They just don't use Python (not usually)
- There is also a mathematical notational that gets used for a similar purpose

Example:

- Semantics of a conditional

(E-IFTRUE) $\texttt{if true then } t_2 \texttt{ else } t_3 \longmapsto t_2$

(E-IFFALSE) $\texttt{if false then } t_2 \texttt{ else } t_3 \longmapsto t_3$

(E-IF)
$$\frac{t_1 \longmapsto t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longmapsto \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$$

- This is defining "small steps"
- Think of it as defining substitutions.

Example: Wasm

Introduction
Structure
Validation
Execution
- Conventions
- Runtime Structure
- Numerics
- Instructions
- Modules

Binary Format

if $blocktype$ $instr_1^*$ else $instr_2^*$ end

1. Assert: due to validation, $\mathrm{expand}_F(blocktype)$ is defined.
2. Let $[t_1^m] \rightarrow [t_2^n]$ be the function type $\mathrm{expand}_F(blocktype)$.
3. Let L be the label whose arity is n and whose continuation is the end of the if instruction.
4. Assert: due to validation, a value of value type i32 is on the top of the stack.
5. Pop the value i32.const c from the stack.
6. Assert: due to validation, there are at least m values on the top of the stack.
7. Pop the values val^m from the stack.
8. If c is non-zero, then:
   a. Enter the block val^m $instr_1^*$ with label L.
9. Else:
   a. Enter the block val^m $instr_2^*$ with label L.

$$F; val^m\ (\mathrm{i32.const}\ c)\ \mathrm{if}\ bt\ instr_1^*\ \mathrm{else}\ instr_2^*\ \mathrm{end} \hookrightarrow F; \mathrm{label}_n\{\epsilon\}\ val^m\ instr_1^*\ \mathrm{end} \quad (\mathrm{if}\ c \neq 0 \wedge \mathrm{expand}_F(bt) = [t_1^m] \rightarrow [t_2^n])$$
$$F; val^m\ (\mathrm{i32.const}\ c)\ \mathrm{if}\ bt\ instr_1^*\ \mathrm{else}\ instr_2^*\ \mathrm{end} \hookrightarrow F; \mathrm{label}_n\{\epsilon\}\ val^m\ instr_2^*\ \mathrm{end} \quad (\mathrm{if}\ c = 0 \wedge \mathrm{expand}_F(bt) = [t_1^m] \rightarrow [t_2^n])$$

- All of this can be very difficult to read (to me)
- Big Picture: it's describing an <u>interpreter</u>

Project

- Find the file `wabbit/interp.py`
- Follow instructions inside.
- Goal: Can we more precisely define/ understand the semantics of Wabbit by writing an interpreter that runs programs directly from the data model?